10. The Database Access API

Created: April 1, 2003
Updated: April 20, 2004

Database Access [Library **dbapi**: include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This library provides the underlying user-layer and driver API for the NCBI database connectivity project. The project's goal is to provide a access to various relational database management systems (RDBMS) with a single uniform user interface. Consult the detailed documentation for details of the supported DBAPI drivers.

Chapter Outline

The following is an outline of the topics presented in this chapter:

- DBAPI Overview
- NCBI DBAPI User-Layer Reference
    - Object hierarchy
    - Includes
    - Objects
    - Object Life Cycle
    - CVariant type
    - Choosing the driver
    - Data Source and Connections
    - Main loop
    - Input and Output Parameters
    - Stored Procedures
    - Cursors

- Updating BLOBs using cursors
- Using bulk insert

- NCBI DBAPI Driver Reference
  - Overview
  - The driver architecture
  - Sample program
  - Topics
    - Error handling
    - Driver context and connections
    - Driver Manager
    - Text and Image Data Handling
    - Results loop
  - Supported DBAPI drivers
    - Sybase CTLIB
    - Sybase DBLIB
    - Microsoft DBLIB
    - FreeTDS 0.60 (TDS ver. 8.0)
    - ODBC
    - MySQL Driver

dbapi [HYPERLINK "http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/include/dbapi"include/dbapi | src/dbapi]

driver [include/dbapi/driver | src/dbapi/driver]

DBAPI Overview

(*pending*: DBAPI motivation, layers and architecture)

NCBI DBAPI User-Layer Reference

Object hierarchy

See Figure 1.

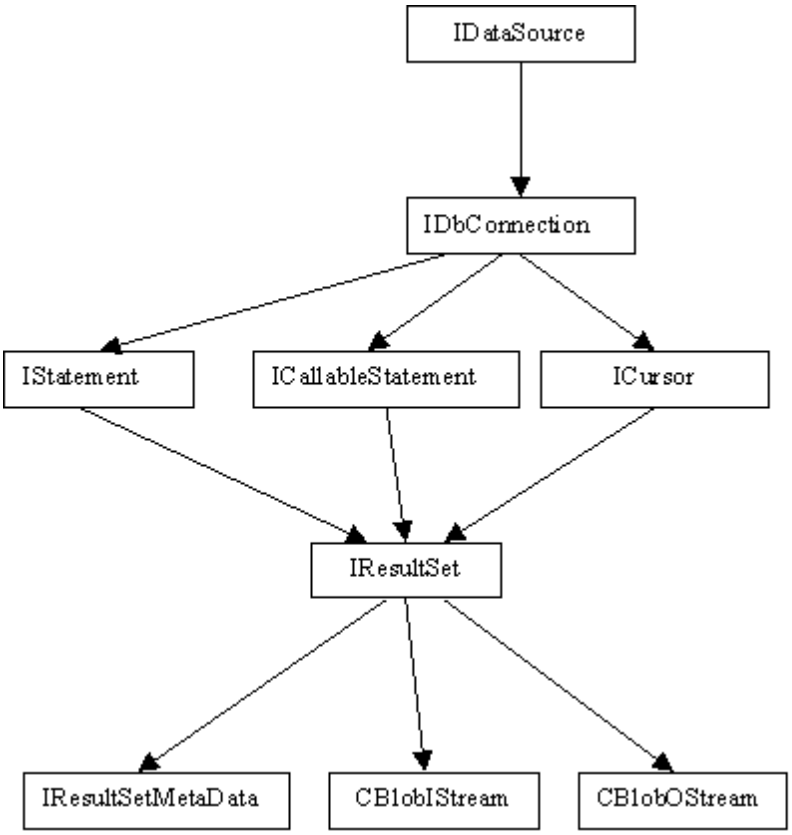

Figure 1: Object Hierarchy

Includes

For most purposes it is sufficient to include one file in the user source file: *dbapi.hpp*.

```
#include <dbapi/dbapi.hpp>
```

For static linkage the following include file is also necessary:

```
#include <dbapi/driver/drivers.hpp>
```

Objects

All objects are returned by pointers to their respective interfaces. The null (0) value is valid, meaning that no object was returned.

Object Life Cycle

In general, any child object is valid only in the scope of its parent object. This is because most of the objects share the same internal structures. There is no need to delete every object explicitly, as all created objects will be deleted upon program exit. Specifically, all objects are derived from the static CDriverManager object, and will be destroyed when CDriverManager is destroyed. It is possible to delete any object from the framework and it is deleted along with all derived objects. For example, when a IConnection object is deleted, all derived IStatement, ICallableStatement and IResultSet objects will be deleted too. For each object a Close() method is provided. It dis-

poses of internal resources, required for the proper library cleanup, but leaves the framework intact. After calling Close() the object becomes invalid. This method may be necessary when the database cleanup and framework cleanup are performed in different places in the code.

CVariant type

The ***CVariant*** type is used to represent any database data type (except BLOBs). It is an object, not a pointer, so it behaves like a primitive C++ type. Basic comparison operators are supported (==, !=, <) for identical internal types. If types are not identical, ***CVariantException*** is thrown. ***CVariant*** has a set of getters to extract a value of a particular type, e.g. ***GetInt4(), GetByte(), GetString()***, etc. If ***GetString()*** is called for a different type, like ***DateTime*** or ***integer*** it tries to convert it to a string. It it doesn't succeed, ***CVariantException*** is thrown. There is a set of factory methods (static functions) for creating ***CVariant*** objects of a particular type, such as ***CVariant::BigInt()***, ***CVariant::SmallDateTime()***, ***CVariant::VarBinary()*** etc. For more details please see the comments in variant.hpp file.

Choosing the driver

There are several drivers for working with different SQL servers on different platforms. The ones presently implemented are "`ctlib`" (Sybase), "`dblib`"(MS SQL, Sybase), "`ftds`" (MS SQLcross platform). For static linkage these drivers should be registered manually; for dynamic linkage this is not necessary. The ***CDriverManager*** object maintains all registered drivers.

```
 CDriverManager &dm = CDriverManager::GetInstance();
DBAPI_RegisterDriver_CTLIB(dm);
DBAPI_RegisterDriver_DBLIB(dm);
```

Data Source and Connections

The ***IDataSource*** interface defines the database platform. To create an object implementing this interface, use the method ***CreateDs(const string& driver)***. An ***IDataSource*** can create objects represented by an ***IConnection*** interface, which is responsible for the connection to the database. It is highly recommended to specify the database name as an argument to the ***CreateConnction()*** method, or use the ***SetDatabase()*** method of a ***CConneciton*** object instead of using a regular SQL statement. In the later case, the library won't be able to track the current database.

```
 IDataSource *ds = dm.CreateDs("ctlib");
IConnection *conn = ds->CreateConnection();
conn->Connect("user", "password", "server", "database");
IStatement *stmt = conn->CreateStatement();
```

Every additional call to ***IConnection::CreateStatement()*** results in cloning the connection for each statement. These connections inherit the same default database, which was specified in the ***Connect()*** or ***SetDatabase()*** method. Thus if the default database was changed by calling ***SetDatabase()***, all subsequent cloned connections created by ***CreateStatement()*** will inherit this particular default database.

Main loop

The library simulates the main result-retrieving loop of the Sybase client library by using the ***IStatement::HasMoreResults()*** method:

```
 stmt->Execute("select à");
while( stmt->HasMoreResults() ) {
    if( stmt->HasRows() ) {
        IResultSet *rs = stmt->GetResultset();

        // Retrieve results, if any

        while( rs->Next() ) {
            int col1 = rs->GetVariant(1).GetInt4();
            ...
        }
    }
}
```

This method should be called until it returns false, which means that no more results is available. It returns as soon as a result is ready. The type of the result can be obtained by calling the ***IResultSet::GetResultType()*** method. Supported result types are `eDB_RowResult, eDB_ParamResult, eDB_ComputeResult, eDB_StatusResult, eDB_CursorResult`. The method ***IStatement::GetRowCount()*** returns the number of updated or deleted rows.

The ***IStatement::ExecuteUpdate()*** method is used for SQL statements that do not return rows:

```
     stmt->ExecuteUpdate("update...");
int rows = stmt->GetRowCount();
```

The method ***IStatement::GetResultSet()*** returns an ***IResultSet*** object. The method ***IResultSet::Next()*** actually does fetch, so it should be always called first. It returns false when no more fetch data is available. All column data, except Image and Text is represented by a single ***CVariant*** object. The method ***IResultSet::GetVariant()*** takes one parameter û column number. Column numbers start with 1.

Input and Output Parameters

The method ***ICallableStatement::SetParam(const CVariant& v, const string& name)*** is used to pass parameters to stored procedures and dynamic SQL statements. To ensure the correct parameter type it is recommended to use CVariant type factories (static methods) to create a CVariant of the required internal type. There is no internal representation for the BIT parameter type, please use TinyInt of Int types with 0 for false and 1 for true respectively. Here are a few examples: CVariant::Int4(Int4 *p), CVariant::TinyInt(UInt1 *p), CVariant::VarChar(const char *p, size_t len) etc.

There are also corresponding constructors, like CVariant::CVariant(Int4 v), CVariant:: CVariant(const string& s), ..., but the user must ensure the proper type conversion in the arguments, and not all internal types can be created using constructors.

Output parameters are set by the ***ICallableStatement::SetOutputParam(const CVariant& v, const string& name)*** method, where the first argument is a null ***CVariant*** of a particular type, e.g. *SetOutputParam(CVariant(eDB_SmallInt),"@arg")*.

Stored Procedures

The ***ICallableStatement*** object is used for calling stored procedures. Fist get the object itself by calling ***IConnection::PrepareCall()***. Then set any parameters. If the parameter name is empty, the calls to ***SetParam()*** should be in the exact order of the actual parameters. Retrieve all results in the main loop. Get the status of the stored procedure using the ***ICallableStatement::GetReturnStatus()*** method.

```
 ICallableStatement *cstmt = conn->PrepareCall("ProcName");
Uint1 byte = 1;
cstmt->SetParam(CVariant("test"), "@test_input");
cstmt->SetParam(CVariant::TinyInt(&byte), "@byte");
cstmt->SetOutputParam(CVariant(eDB_Int), "@result");
cstmt->Execute();
while(cstmt->HasMoreResults()) {
    if( cstmt->HasRows() ) {
        IResultSet *rs = cstmt->GetResultSet();
        switch( rs->GetResultType() ) {
            case eDB_RowResult:
                while(rs->Next()) {

                // retrieve row results

                }
                break;
            case eDB_ParamResult:
                while(rs->Next()) {

                // Retrieve parameter row

                }
                break;
        }
    }
}

// Get status
int status = cstmt->GetReturnStatus();
```

It is also possible to use ***IStatement*** interface to call stored procedures using standard SQL language call. The difference from ***ICallableStatement*** is that there is no ***-SetOutputParam()***-call. The output parameter is passed as regular ***-SetParam()***- call with *non null****CVariant*** argument. There is no ***GetReturnStatus()*** call in ***IStatement*** , so use the result type filter to get it.

```
     sql = "exec SampleProc @id, @f, @o output";
stmt->SetParam(CVariant(5), "@id");
stmt->SetParam(CVariant::Float(&f), "@f");
stmt->SetParam(CVariant(5), "@o");
stmt->Execute(sql);
```

```
while(stmt->HasMoreResults()) {
    IResultSet *rs = stmt->GetResultSet();

    if( rs == 0 )
        continue;

    switch( rs->GetResultType() ) {
    case eDB_ParamResult:
        while( rs->Next() ) {
            NcbiCout << "Output param: "
                     << rs->GetVariant(1).GetInt4()
                     << endl;
        }
        break;
    case eDB_StatusResult:
        while( rs->Next() ) {
            NcbiCout << "Return status: "
                     << rs->GetVariant(1).GetInt4()
                     << endl;
        }
        break;
    case eDB_RowResult:
        while( rs->Next() ) {
            if( rs->GetVariant(1).GetInt4() == 2121 ) {
                NcbiCout << rs->GetVariant(2).GetString() << "|"
                         << rs->GetVariant(3).GetString() << "|"
                         << rs->GetVariant(4).GetString() << "|"
                         << rs->GetVariant(5).GetString() << "|"
                         << rs->GetVariant(6).GetString() << "|"
                         << rs->GetVariant(7).GetString() << "|"
                         << endl;
            } else {
                NcbiCout << rs->GetVariant(1).GetInt4() << "|"
                         << rs->GetVariant(2).GetFloat() << "|"
                         << rs->GetVariant("date_val").GetString() << "|"
                         << endl;
            }
        }
        break;
    }
}
stmt->ClearParamList();
```

Cursors

The library currently supports basic cursor features such as setting parameters and cursor update and delete operations.

```
 ICursor *cur = conn->CreateCursor("table_cur",
                                   "select ... for update of ...");
IResultSet *rs = cur->Open();
while(rs->Next()) {
```

```
    cur->Update(table, sql_statement_for_update);
}
cur->Close();
```

Updating BLOBs using cursors

It is recommended to update BLOBs using cursors, because this is the only way to work with ODBC driver and no additional connection is open.

```
 ICursor *blobCur = conn->CreateCursor("test",
             "select id, blob from BlobSample for update of blob");
IResultSet *blobRs = blobCur->Open();
while(blobRs->Next()) {
    ostream& out = blobCur->GetBlobOStream(2, blob.size());
    out.write(buf, blob.size());
    out.flush();
}
```

Note that GetBlobOStream() takes the column number as the first argument and this call is invalid untill the cursor is open.

Using bulk insert

Bulk insert is useful when it is necessary to insert big amounts of data. The ***IConnection::CreateBulkInsert()*** takes two parameters, the table name and number of columns. The ***CVariant::Truncate(size_t len)*** method truncates the internal buffer of CDB_Text and CDB_Image object from the end of the buffer. If no paramter specified, it erases the whole buffer.

```
NcbiCout << "Initializing BlobSample table..." << endl;
string im = ......;
IBulkInsert *bi = conn->CreateBulkInsert("BlobSample", 2);
CVariant col1 = CVariant(eDB_Int);
CVariant col2 = CVariant(eDB_Text);
bi->Bind(1, &col1);
bi->Bind(2, &col2);
for(int i = 0; i < ROWCOUNT; ++i ) {
    string im = "BLOB data " + NStr::IntToString(i);
    col1 = i;
    col2.Truncate();
    col2.Append(im.c_str(), im.size());
    bi->AddRow();
}
bi->Complete();
```

NCBI DBAPI Driver Reference

(Low-level access to the various RDBMSs.)

- Overview
- The driver architecture

- Sample program
- Topics
    - Error handling
    - Driver context and connections
    - Driver Manager
    - Text and Image Data Handling
    - Results loop
- Supported DBAPI drivers
    - Sybase CTLIB
    - Sybase DBLIB
    - Microsoft DBLIB
    - FreeTDS 0.60 (TDS ver. 8.0)
    - ODBC
    - MySQL Driver

Overview

The *NCBI DBAPI driver library* describes and implements a set of objects needed to provide a uniform low-level access to the various relational database management systems (RDBMS). The basic driver functionality is the same as in most other RDBMS client APIs. It allows to open a connection to a server, execute a command (query) on this connection and get the results back. The main advantage of using the driver is that you don't have to change your own upper-level code if you need to move from one RDBMS client API to another.

The driver can use two different methods to access the particular RDBMS. If RDBMS provides a client library for a given computer system (i.e. Sun/Solaris), then driver utilizes this library. If there is no client library, then driver connects to RDBMS through a special *gateway server* which is running on a computer system where such library does exist.

The driver architecture

There are two major groups of the driver's objects: the RDBMS independent objects, and the objects which are specific to a particular RDBMS. The only RDBMS specific object which user should be aware of is a "Driver Context". The "Driver Context" is effectively a "Connection" fac-

tory. The only way to make a connection to the server is to call the ***Connect()*** method of a "Driver Context" object. So, before doing anything with RDBMS, you need to create at least one driver context object. All driver contexts implement the same interface defined in I_DriverContext class. If you are working on a library which could be used with various RDBMS it is a good idea do not create the driver context inside the library, but take a pointer to I_DriverContext instead.

There is no "real" factory for the driver contexts. The reason for that is it's not always possible to statically link in the same binary the RDBMS libraries from different vendors. Most of them are written in C and the name collisions do exist. The Driver Manager helps to overcome this problem. It allowes to create a mixture of statically linked and dynamically loaded drivers and use them together in one executable.

The driver context creates the connection which is RDBMS specific, but before returning it to the caller it puts it into an "envelope" of RDBMS independent object CDB_Connection. The same is true for the commands and for the results - user gets the pointer to RDBMS independent "envelope object" instead of the real one. This is a caller responsibility to delete those objects. The life spans of the real object and the envelope one are not necessarily the same.

Once you have got the connection object, you could use it as a factory for the different types of commands. The command object in it's turn serves as a factory for the results. The connection is always single threaded, that means that you have to execute the commands and process their results sequentially one by one. If you need to execute the several commands in parallel, you can do it using multiple connections.

Another important part of the driver is an error and message handling. There are two different mechanisms implemented. The first one is exceptions. All exceptions which could be thrown by the driver are inherited from the single base class CDB_Exception. Driver uses the exception mechanism whenever it's possible, but in many cases the underlying client library uses the callbacks or handlers to report the error messages and prevents from throwing the exceptions. The driver supply a handler's stack mechanism to manage these cases.

To send and to receive the data through the driver you have to use the driver provided datatypes. The collection of the datatypes includes: one, two, four and eight byte integers; float and double; numeric; char, varchar, binary, varbinary; datetime and smalldatetime; text and image. All datatypes are derived from a single base class CDB_Object.

Sample program

This program opens one connection to the server and selects the database names and the date when each database was created (assuming that table "sysdatabases" does exist). In this example the string "XXX" should be replaced with the real driver name.

```
 #include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
/* Here, XXXlib has to be replaced with the real name, e.g. "ctlib" */
#include <dbapi/driver/XXXlib/interfaces.hpp>
USING_NCBI_SCOPE;
int main()
{
    try { // to be sure that we are catching all driver related exceptions
```

```
        // We need to create a driver context first
        // In real program we have to replace CXXXContext with something real
        CXXXContext my_context;
        // connecting to server "MyServer"
        // with user name "my_user_name" and password "my_password"
        CDB_Connection* con = my_context.Connect("MyServer", "my_user_name",
                                                 "my_password", 0);
        // Preparing a SQL query
        CDB_LangCmd* lcmd =
            con->LangCmd("select name, crdate from sysdatabases");
        // Sending this query to a server
        lcmd->Send();
        CDB_Char dbname(64);
        CDB_DateTime crdate;
        // the result loop
        while(lcmd->HasMoreResults()) {
            CDB_Result* r= lcmd->Result();
            // skip all but row result
            if (r == 0  ||  r->ResultType() != eDB_RowResult) {
                delete r;
                continue;
            }
            // printing the names of selected columns
            cout << r->ItemName(0) << " \t\t\t"
                 << r->ItemName(1) << endl;
            // fetching the rows
            while ( r->Fetch() ) {
                r->GetItem(&dbname); // get the database name
                r->GetItem(&crdate); // get the creation date
                cout << dbname.Value() << ' '
                     << crdate.Value().AsString("M/D/Y h:m")
                     << endl;
            }
            delete r; // we don't need this result anymore
        }
        delete lcmd; // delete the command
        delete con;  // delete the connection
    }
    catch (CDB_Exception& e) { // printing the error messages
        CDB_UserHandler_Stream myExHandler(&cerr);
        myExHandler.HandleIt(&e);
    }
}
```

Topics

Error handling

The error handling is almost always a pain when you are working with RDBMS. The different systems implement the different approaches. You could get the error messages through the return codes, callbacks, handlers and/or exceptions. These messages could have different formats. It could be just an integer (error code) or some structure or a set of callback's arguments. The *NCBI*

DBAPI driver intercepts all those error messages in all different formats and converts them into the objects of CDB_Exception derived types. The following types are used: CDB_SQLEx This type is used if error message has come from a SQL server and indicates an error in SQL query. It could be a wrong table or column name or just a wrong syntax of SQL query. The message details could be obtained using the following methods:

- ***OriginatedFrom()*** - returns a SQL server name
- ***BatchLine()*** - returns a line number in SQL batch which did generate an error
- ***Message()*** - returns the error message itself
- ***Severity()*** - returns the severity of this message (assigned by SQL server)
- ***ErrCode()*** - returns the integer code for this message (assigned by SQL server)
- ***SqlState()*** - returns a byte string describing an error (it's not useful most of the time)

CDB_RPCEx An error message has come while RPC or stored procedure was executed on a server. The methods to use:

- ***OriginatedFrom()*** - returns a server name
- ***ProcName()*** - returns a procedure name
- ***ProcLine()*** - returns a line number inside the procedure
- ***Message()*** - returns the error message itself
- ***Severity()*** - returns the severity of this message (assigned by a server)
- ***ErrCode()*** - returns the integer code for this message (assigned by a server)

CDB_DeadlockEx To report about deadlock. The methods to use:

- ***OriginatedFrom()*** - returns a SQL server name
- ***Message()*** - returns the error message itself

CDB_DSEx Any error which has come from a RDBMS and is not a SQL query or RPC related. The methods to use:

- ***OriginatedFrom()*** - returns a server name
- ***Message()*** - returns the error message itself

- ***Severity()*** - returns the severity of this message (assigned by a server)
- ***ErrCode()*** - returns the integer code for this message (assigned by a server)

CDB_TimeoutEx To report about timeout. The methods to use:

- ***OriginatedFrom()*** - returns a server name
- ***Message()*** - returns the error message itself

CDB_ClientEx Any client side error. The methods to use:

- ***OriginatedFrom()*** - returns the name of method or function which reports the error
- ***Message()*** - returns the error message itself
- ***Severity()*** - returns the severity of this message
- ***ErrCode()*** - returns the integer code for this message

Driver uses two ways to deliver the error message object to an application. If it is possible to throw an exception, then driver throws the error message object. If not, then driver calls the user's error handler with a pointer to this object as an argument. It's not always convenient to process all types of error messages in one error handler. Some users may want to use a special error message handler inside some function or block and a default error handler outside. To accommodate these cases the driver provides a handler stack mechanism. The top handler in the stack gets the error message object first. If it knows how to deal with this message, then it processes the message and returns *true*. If handler wants to pass this message to the other handlers, then it returns *false*. So, driver pushing the error message object through the stack until it gets true from the handler. The default driver's error handler which just printout the error message to stderr is always on a bottom of the stack. The another tool which user may want to use for error handling is the CDB_MultiEx objects. This tool allows to collect the multiple CDB_Exception objects into one container and than **throw** this container as one object.

Driver context and connections

Every program which is going to work with *NCBI DBAPI driver* should create at least one Driver Context object first. The main purpose of this object is to be a Connection factory, but it's a good idea to customize this object a little bit prior to open any connection. The first step is to setup two message handler stacks. The first one is for error messages which are not bound to some particular connection or could occur inside the ***Connect()*** method. Use ***PushCntxMsgHandler()*** to populate it. The other stack serves as a initial message handler stack for all connections which will be derived from this context. Use ***PushDefConnMsgHandler()*** method to populate this stack. The second step of customization is a time-outs setting. The ***SetLoginTimeout()*** and ***SetTime-***

out() methods do the job. If you are going to work with text or image objects in your program, you need to call ***SetMaxTextImageSize()*** to define the maximal size for such objects. Objects which exceed this limit could be truncated.

```
 class CMyHandlerForConnectionBoundErrors : public CDB_UserHandler
{
    virtual bool HandleIt(CDB_Exception* ex);
    ...
};
class CMyHandlerForOtherErrors : public CDB_UserHandler
{
    virtual bool HandleIt(CDB_Exception* ex);
    ...
};
...
int main()
{
    CMyHandlerForConnectionBoundErrors conn_handler;
    CMyHandlerForOtherErrors           other_handler;
    ...
    try { // to be sure that we are catching all driver related exceptions
        // We need to create a driver context first
        // In real program we have to replace CXXXContext with something real
        CXXXContext my_context;
        my_context.PushCntxMsgHandler(&other_handler);
        my_context.PushDefConnMsgHandler(&conn_handler);
        // set timeouts (in seconds) and size limits (in bytes):
        my_context.SetLoginTimeout(10); // for logins
        my_context.SetTimeout(15);      // for client/server communications
        my_context.SetMaxTextImageSize(0x7FFFFFFF); // text/image size limit
        ...
        CDB_Connection* my_con =
            my_context.Connect("MyServer", "my_user_name", "my_password",
                               I_DriverContext::fBcpIn);
        ...
    }
    catch (CDB_Exception& e) {
        other_handler.HandleIt(&e);
    }
}
```

The only way to get a connection to a server in *NCBI DBAPI driver* is through a ***Connect()*** method in driver context. The first three arguments: server name, user name and password are obvious. Values for `mode` are constructed by a bitwise-inclusive-OR of flags defined in EConnectionMode. If `reusable` is *false*, then driver creates a new connection which will be destroyed as soon as user delete the correspondent CDB_Connection (the `pool_name` is ignored in this case).

Opening a connection to a server is an expensive operation. If program opens and closes connections to the same server multiple times it worth to call the ***Connect()*** method with `reusable` set to *true*. In this case driver does not close the connection when the correspondent

CDB_Connection is deleted, but keeps it around in a "recycle bin". Every time an application calls the ***Connect()*** method with `reusable` set to *true*, driver tries to satisfy the request from a "recycle bin" first and opens a new connection only if it is necessary.

The `pool_name` argument is just an arbitrary string. Application could use this argument to assign a name to one or more connections (to create a connection pool) or to invoke a connection by name from this pool.

```
...
// Create a pool of four connections (two to one server and two to another)
// with the default database "DatabaseA"
CDB_Connection* con[4];
int i;
for (i = 4;  i--; ) {
    con[i]= my_context.Connect((i%2 == 0) ? "MyServer1" : "MyServer2",
                               "my_user_name", "my_password", 0, true,
                               "ConnectionPoolA");
    CDB_LangCmd* lcmd= con[i]->LangCmd("use DatabaseA");
    lcmd->Send();
    while(lcmd->HasMoreResults()) {
        CDB_Result* r = lcmd->Result();
        delete r;
    }
    delete lcmd;
}
// return all connections to a "recycle bin"
for(i= 0; i < 4; delete con_array[i++]);
...
// in some other part of the program
// we want to get a connection from "ConnectionPoolA"
// but we don't want driver to open a new connection if pool is empty
try {
    CDB_Connection* my_con= my_context.Connect("", "", "", 0, true,
                                               "ConnectionPoolA");
    // Note that server name, user name and password are empty
    ...
}
catch (CDB_Exception& e) {
    // the pool is empty
    ...
}
```

Application could combine in one pool the connections to the different servers. This mechanism could also be used to group together the connections with some particular settings (default database, transaction isolation level, etc.).

Driver Manager

It's not always known upfront which *NCBI DBAPI driver* will be used in some particular program. Sometimes you want a driver to be a parameter in your program. Sometimes you need to use two different drivers in one binary but can not link them statically because of name collisions. Sometimes you just need the driver contexts factory. The Driver Manager is intended to solve these problems.

Let's rewrite our Sample program using the *Driver Manager*. The original text was.

```
 #include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
/* Here, XXXlib has to be replaced with the real name, e.g. "ctlib" */
#include <dbapi/driver/XXXlib/interfaces.hpp>
USING_NCBI_SCOPE;
int main()
{
    try { // to be sure that we are catching all driver related exceptions
        // We need to create a driver context first
        // In real program we have to replace CXXXContext with something real
        CXXXContext my_context;
        // connecting to server "MyServer"
        // with user name "my_user_name" and password "my_password"
        CDB_Connection* con = my_context.Connect("MyServer", "my_user_name",
                                                 "my_password", 0);
        ...
```

If we use the *Driver Manager* we could allow the driver name to be a program argument.

```
 #include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
#include <dbapi/driver/driver_mgr.hpp> // this is a new header
USING_NCBI_SCOPE;
int main(int argc, const char* argv[])
{
    try { // to be sure that we are catching all driver related exceptions
        C_DriverMgr drv_mgr;
        // We need to create a driver context first
        I_DriverContext* my_context= drv_mgr.GetDriverContext(
                                       (argc > 1)? argv[1] : "ctlib");
        // connecting to server "MyServer"
        // with user name "my_user_name" and password "my_password"
        CDB_Connection* con = my_context->Connect("MyServer", "my_user_name",
                                                  "my_password", 0);
        ...
```

This fragment creates an instance of *Driver Manager*, dynamically loads driver's library, implicitly register this driver, creates the driver context and makes a connection to a server. If you don't want to load some drivers dynamically for any reason, but want to use the *Driver Manager* as a driver contexts factory, then you need to statically link your program with those libraries and explicitly registered them using functions from *dbapi/driver/drivers.hpp* header.

Text and Image Data Handling

The ***text*** and ***image*** are SQL datatypes which can hold up to 2Gb of data. Because they could be huge, RDBMS keep these values separately from the other data in the table. In most cases the table itself keeps just a special pointer to a text/image value and an actual value occupies a separate disk space. This implicates some difficulties in text/image data handling.

When you retrieves a large text/image value, you often prefer to "stream" it into your program and process it chunk by chunk rather than get it as one piece. Some RDBMS clients allow to stream the text/image values only if a correspondent column is the only column in select statement.

Let's suppose that you do have a table: table T (i_val int, t_val text) And you need to select all i_val, t_val where i_val > 0. The simplest way is to use a query:

```
select i_val, t_val from T where i_val > 0
```

But it could be expensive. Because two columns are selected, some clients will put the whole row in a buffer prior to give the access to it to the user. The better way to do this is to use two selects:

```
select i_val from T where i_val > 0select t_val from T where i_val > 0
```

Looks ugly, but could save you a lot of memory.

Updating and inserting the text/image data is also not a straightforward process. For small texts and images it is possible to use just SQL *insert* and *update* statements, but it will be inefficient (if possible at all) for the large ones. The better ways to insert and to update the texts and images is to use ***SendData()*** method of CDB_Connection object or to use the CDB_SendDataCmd object.

Recommended algorithm for inserting the text/image data:

- Using a SQL *insert* statement insert a new row into a table. Use *""* value for each ***text*** column (*0x0* for *image* column) you are going to populate. Use *NULL* only if this value is going to remain *NULL*.
- Using a SQL *select* statement select all text/image columns from this row.
- Fetch the row result and get a I_ITDescriptor for each column
- Finish the results loop
- Use ***SendData()*** method or CDB_SendDataCmd object to populate the columns.

Example

Let's suppose that we want to insert a new row into table T described above.

```
 CDB_Connection* con;
...
// preparing the query
CDB_LangCmd* lcmd= con->LangCmd("insert T (i_val, t_val) values(100, ' ')\n");
lcmd->More("select t_val from T where i_val = 100");
// Sending this query to a server
```

```
lcmd->Send();
I_ITDescriptor* my_descr;
// the result loop
while(lcmd->HasMoreResults()) {
    CDB_Result* r= lcmd->Result();
    // skip all but row result
    if (r == 0  ||  r->ResultType() != eDB_RowResult) {
        delete r;
        continue;
    }
    // fetching the row
    while(r->Fetch()) {
        // read 0 bytes from the text (some clients need this trick)
        r->ReadItem(0, 0);
        my_deskr = r->GetImageOrTextDescriptor();
    }
    delete r; // we don't need this result anymore
}
delete lcmd; // delete the command
CDB_Text my_text;
my_text.Append("This is a text I want to insert");
//sending the text
con->SendData(my_descr, my_text);
delete my_descr; // we don't need this descriptor anymore
...
```

Recommended algorithm for updating the text/image data:

- Using a SQL *update* statement replace the current value with *""* for text column (*0x0* for image)
- Using a SQL *select* statement select all text/image columns you want to update in this row.
- Fetch the row result and get a I_ITDescriptor for each column
- Finish the results loop
- Use ***SendData()*** method or CDB_SendDataCmd object to populate the columns.

Example

```
 CDB_Connection* con;
...
// preparing the query
CDB_LangCmd* lcmd= con->LangCmd("update T set t_val= ' ' where i_val =  100");
lcmd->More("select t_val from T where i_val = 100");
// Sending this query to a server
lcmd->Send();
I_ITDescriptor* my_descr;
// the result loop
while(lcmd->HasMoreResults()) {
    CDB_Result* r= lcmd->Result();
    // skip all but row result
```

```
        if (r == 0  ||  r->ResultType() != eDB_RowResult) {
            delete r;
            continue;
        }
        // fetching the row
        while(r->Fetch()) {
            // read 0 bytes from the text (some clients need this trick)
            r->ReadItem(0, 0);
            my_deskr = r->GetImageOrTextDescriptor();
        }
        delete r; // we don't need this result anymore
    }
    delete lcmd; // delete the command
    CDB_Text my_text;
    my_text.Append("This is a text I want to see as an update");
    //sending the text
    con->SendData(my_descr, my_text);
    delete my_descr; // we don't need this descriptor anymore
    ...
```

Results loop

The connection in *NCBI DBAPI driver* is always single threaded. Application has to retrieve all results from a current command prior to executing a new one. Not all of the results are always meaningful for the application (i.e. an RPC always returns a status result regardles of either a procedure has a "**return** something" statement or not), but all of them need to be retrieved. The following results loop is recommended for all types of the commands:

```
 CDB_XXXCmd* cmd; // XXX could be Lang, RPC, etc.
...
while (cmd->HasMoreResults()) {
        // HasMoreResults() method returns true        // if the Result() method needs to
be called.
        // It doesn't guarantee that Result() will return not NULL result
   CDB_Result* res = cmd->Result();
   if (res == 0)
       continue; // a NULL res doesn't mean that there is no more results
   switch(res->ResultType()) {
      case eDB_RowResult: // row result
          while(res->Fetch()) {
             ...
          }
          break;
      case eDB_ParamResult: // Output parameters
          while(res->Fetch()) {
             ...
          }
          break;
      case eDB_ComputeResult: // Compute result
          while(res->Fetch()) {
             ...
          }
          break;
      case eDB_StatusResult: // Status result
```

```
                while(res->Fetch()) {
                   ...
                }
                break;
            case eDB_CursorResult: // Cursor result
                while(res->Fetch()) {
                   ...
                }
                break;
        }
        delete res;
    }
```

If you don't want to process some particular type of result, just skip the *while (res->Fetch()) {...}* in the corresponding *case*.

Supported DBAPI drivers

- Sybase CTLIB
- Sybase DBLIB
- Microsoft DBLIB
- FreeTDS 0.60 (TDS ver. 8.0)
- ODBC
- MySQL Driver

Sybase CTLIB

- Registration function (for the manual, static registration) ***DBAPI_RegisterDriver_CTLIB()***
- Driver default name (for the run-time loading from a DLL) *"ctlib"*
- Driver library `dbapi_driver_ctlib`
- *Sybase CTLIB* libraries and headers used by the driver (UNIX) `$(SYBASE_LIBS)` `$(SYBASE_INCLUDE)`
- *Sybase CTLIB* libraries and headers used by the driver (MS Windows) You will need Sybase OpenClient package installed on your PC.Libraries: LIBCT.LIB LIBCS.LIB LIBBLK.LIB. In MSVC++, go to *"Tools" / "Options..." / "Directories"* and set up the path to Sybase OpenClient libraries and headers (for example "C:\Sybase\lib" and "C:\Sybase\include" respectively). To run the application, you must set environment variable %SYBASE% to

the Sybase OpenClient root directory (e.g. "C:\Sybase"), and also to have your interface file there, in INI/sql.ini. In NCBI, we have the Sybase OpenClient libs installed in \\DIZZY \public\Sybase.

- CTLIB-specific header (contains *non-portable* extensions) *dbapi/driver/ctlib/interfaces.hpp*
- CTLIB-specific driver context attributes "reuse_context", default = *"true"* "version", default = *"110"* (also allowed:*"100"*)
- Caveats

1. Cannot communicate with MS SQL server using any TDS version.

Sybase DBLIB

- Registration function (for the manual, static registration) ***DBAPI_RegisterDriver_DBLIB()***
- Driver default name (for the run-time loading from a DLL) *"dblib"*
- Driver library `dbapi_driver_dblib`
- *Sybase DBLIB* libraries and headers used by the driver (UNIX) `$(SYBASE_DBLIBS)` `$(SYBASE_INCLUDE)`
- *Sybase DBLIB* libraries and headers used by the driver (MS Windows) Libraries: LIB-SYBDB.LIB See Sybase OpenClient installation and usage instructions in the Sybase CTLIB section (just above).
- DBLIB-specific header (contains *non-portable* extensions) *dbapi/driver/dblib/interfaces.hpp*
- DBLIB-specific driver context attributes "version", default = *"46"* (also allowed:*"100"*)
- Caveats

1. Text/image operations fail when working with MS SQL server, because MS SQL server sends text/image length in the reverse byte order, and this cannot be fixed (as it was fixed for FreeTDS) as we do not have access to the DBLIB source code.
2. DB Library version level *"100"* is recommended for communication with Sybase server 12.5, because the default version level (*"46"*) is not working correctly with this server.

Microsoft DBLIB

- Registration function (for the manual, static registration) ***DBAPI_RegisterDriver_MSDBLIB()***
- Driver default name (for the run-time loading from a DLL) `"msdblib"`
- Driver library `dbapi_driver_msdblib`
- *Microsoft DBLIB* libraries and headers used by the driver `NTWDBLIB.LIB`
- *Microsoft DBLIB*-specific header (contains *non-portable* extensions) *dbapi/driver/msdblib/interfaces.hpp*
- *Microsoft DBLIB*-specific driver context attributes NONE
- Caveats

1. On the lower level, the reading of a blob (image or text) cannot be performed in a pure stream-wise fashion, and therefore the whole blob has to be read in advance. Actually all contents of all columns get read up completely as soon as the row is fetched(!). Although this is hidden from the user code, however it theoretically can cause memory exhaustion and at least some performance overhead if the blob is too big.

FreeTDS 0.60 (TDS ver. 8.0)

- Registration function (for the manual, static registration) ***DBAPI_RegisterDriver_FTDS8()*** ***DBAPI_RegisterDriver_FTDS()***
- Driver default name (for the run-time loading from a DLL) `"ftds"`
- Driver library `dbapi_driver_ftds`
- *FreeTDS* libraries and headers used by the driver `$(FTDS8_LIBS)$(FTDS8_INCLUDE)`
- *FreeTDS*-specific header (contains *non-portable* extensions) *dbapi/driver/ftds/interfaces.hpp*
- *FreeTDS*-specific driver context attributes "version", default = *<DBVERSION_UNKNOWN>* (also allowed: *"42"*, *"46"*, **"70"**, **"80"**, **"100"**)
- Caveats

1. Nobody has ever tried working with any TDS version but the default (<DBVERSION_UNKNOWN>) one.

2. Although a slightly modified version of FreeTDS is now part of the public toolkit, it retains its own license: the GNU Library General Public License.

3. The "compute results" functionality (like from SELECT ... AVERAGE ...) does not work because current FreeTDS implementation cannot decipher the "compute results" specific result set returned by server.

4. RPC is implemened via a language call, so it will work only if the OpenServer it is communicating with has language handler installed (and it is not installed on some NCBI OpenServers!).

5. The FreeTDS client library (the one using TDS protocol version 8.0) was tweaked to work with the MS SQL server and significantly optimized. However, it will not work with Sybase server.

6. Another, earlier non-tweaked version of FreeTDS client library theoretically should be able to work with both MS SQL and SYBASE servers (using TDS protocol version 4.2), however it was not thoroughly tested and can be pretty slow.

ODBC

- Registration function (for the manual, static registration) ***DBAPI_RegisterDriver_ODBC()***
- Driver default name (for the run-time loading from a DLL) *"odbc"*
- Driver library `dbapi_driver_odbc`
- *ODBC* libraries and headers used by the driver (MS Windows) `ODBC32.LIB ODBCCP32.LIB ODBCBCP.LIB`
- *ODBC* libraries and headers used by the driver (UNIX) `$(ODBC_LIBS) $(ODBC_INCLUDE)`
- *ODBC*-specific header (contains *non-portable* extensions) *dbapi/driver/odbc/interfaces.hpp*
- *ODBC*-specific driver context attributes "version", default = *"3"* (also allowed:*"2"*)"use_dsn", default = *"false"* (if you have set this attribute to **"true"**, you need to define your *data source* using *"Control Panel"/"Administrative Tools"/"Data Sources (ODBC)"*)
- Caveats

  1. The ***CDB_Result::GetImageOrTextDescriptor()*** does not work for ODBC driver. You need to use CDB_ITDescriptor instead. The other way to deal with ***texts/images*** in ODBC is through the CDB_CursorCmd methods: ***UpdateTextImage*** and ***SendDataCmd***.

2. On most NCBI PCs, there is an old header *odbcss.h* (from 4/24/1998) installed. The symptom is that although everything compiles just fine, however in the linking stage there are dozens of unresolved symbol errors for ODBC functions. Ask "pc. systems" to fix this for your PC.

3. On UNIX, it's only known to work with Merant's implementation of ODBC, and it has not been thoroughly tested or widely used, so surprises are possible.

MySQL Driver

There is a direct (without ODBC) MySQL driver in the NCBIC++ Toolkit DBAPI. However, the driver realizes a very minimum functionality and does not support the following:

- Working with images by chunks (images can be accessed as string fieldsthough)
- RPC
- BCP
- SendData functionality
- Connection pools
- Parameter binding
- Canceling results
- ReadItem
- IsAlive
- Refresh functions
- Setting timeouts